A Scalable and Loop-Free Multicast Internet Protocol

M. Parsa and J.J. Garcia-Luna-Aceves
Computer Engineering Department, UCSC, Santa Cruz, CA 95064

ABSTRACT

In network multimedia applications, such as multiparty teleconferencing, users often need to send the same information to several (but not necessarily all) other users. To manage such one-to-many or many-to-many communication efficiently in wide-area internetworks, it is imperative to support and perform multicast routing. Multicast routing sends a single copy of a message from a source to multiple receivers over a communication link that is shared by the paths to the receivers. Loop-freedom is a specially important consideration in multicasting. Because applications using multicasting tend to be multimedia and bandwidth intensive, and loops in multicast routing duplicates looping packets.

We present a new multicast routing protocol, called Multicast Internet Protocol (MIP), which offers a simple and flexible approach to constructing both group-shared and shortest-paths multicast trees. MIP can be sender-initiated or receiver-initiated or both; therefore, it can be tailored to the particular nature of an application's group dynamics and size. MIP is independent of the underlying unicast routing algorithms used. MIP is robust and adapts under dynamic network conditions (topology or link cost changes) to maintain loop-free multicast routing. Under stable network conditions, MIP has no maintenance or control message overhead.
Keywords: multicast routing, distributed algorithms

1 INTRODUCTION

To manage one-to-many or many-to-many communication efficiently in wide-area internetworks, it is imperative to support and perform multicast routing. Multicast routing sends only a single copy of a message from a source to multiple receivers over a communication link that is shared by the paths to the receivers. Multicasting can benefit a wide array of network applications, including multiparty video or audio teleconferencing, collaborative environments, replicated databases, resource discovery, and parallel processing.

Multicasting is supported in local area networks (LANs) using hardware technologies. Recently, multicasting has been extended to internetworks by Deering [4]. Based on Deering's work and built into the TCP/IP protocol suite, the internet group message protocol (IGMP) is used to disseminate multicast membership information to multicast routers. Deering's method permits routers to dynamically determine how to forward messages. A delivery tree is constructed on-demand and is data-driven. The tree in the existing IP architecture is the reverse shortest-path tree and shortest-path tree from the source to the group for distance-vector (i.e., DVMRP [11]) and link-state routing (i.e., MOSPF [9]), respectively. For example, the Multicast Backbone (MBone) in today's Internet consists of a set of routers running DVMRP. However, there are several shortcomings with the existing IP multicast architecture, i.e., DVMRP and MOSPF. First, all routers in the Internet have to generate and process periodically control messages for *every* multicast group, regardless of whether or not they belong to the multicast tree of the group. Thus, routers not on the multicast tree incur memory and processing overhead to construct and maintain the tree for the lifetime of the group. Packets that do not lead to any receivers or sources are periodically flooded throughout the Internet, thereby consuming and wasting bandwidth. In DVMRP, it is the data packets that are periodically flooded when the state information for a multicast tree times out. In MOSPF, it is the link-state packets, containing the state information for group membership, that are periodically flooded. Second, the multicast routing information in each router is stored for each source sending to a group. If there are S sources and G groups, the multicast protocols scale as $\Theta(SG)$. Finally, the IP multicast protocols, being extensions of unicast routings, are tightly coupled to the underlying unicast routing algorithm. This complicates inter-domain multicasting if the domains involved use different unicast routing. The unicast routing also becomes more complicated by incorporating the multicast-related requirements.

To overcome the above shortcomings, two protocols have been recently proposed: the core-based tree (CBT) architecture [1] and the protocol independent multicast (PIM) architecture [5]. Although both approaches consti-

Email address: M.P.: courant@cse.ucsc.edu; J.J.G.: jj@cse.ucsc.edu.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1997		**00-00-1997 to 00-00-1997**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Scalable and Loop-Free Multicast Internet Protocol	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Santa Cruz,Department of Computer Engineering,Santa Cruz,CA,95064**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**15**	

tute a substantial improvement over the current multicast architecture, each protocol has its own limitations [10]. A major limitation of CBT is that it constructs only a single tree per group and thus provides longer end-to-end delays than would be obtained along a shortest-path tree. An important limitation of PIM is that the periodic control messages, i.e., its soft-state mechanism, incur overhead even in a stable internet. Both protocols have been shown to suffer from temporary loops resulting from the use of inconsistent unicast routing information [10], and neither protocol has been verified to provide correct multicast routing trees after network changes.

We present a new multicast routing protocol called *Multicast Internet Protocol* (MIP), which solves the shortcomings of the previous approaches to multicast routing. MIP offers a simple and flexible approach to the construction of both group-shared and shortest-path multicast trees. The shortest-path trees in MIP can be relaxed to cost-bounded trees, making it possible to trade off optimality of the tree with the control message overhead of maintaining a shortest-path tree. MIP accommodates sender-initiated and receiver-initiated multicast tree construction, which makes MIP flexible to use in a wide range of applications with different characteristics, group dynamics and sizes. Moreover, these two modes of tree construction are interoperable. MIP is independent of the underlying unicast routing. MIP never creates loops in a multicast tree, even when the underlying unicast routing tables are inconsistent and contain routing-table loops. Loop-freedom is important in multicasting, looping causes packets to multiply each time they traverse a loop. The bouncing-effect and counting-to-infinity problems in the presence of loops prolong the adverse effects on the message overhead of non-loop-free routing protocols. Instead of using the idea of "soft state" to maintain multicast routing information, MIP uses diffusing computations to update and disseminate multicast routing information. MIP's use of diffusing computations insures loop-freedom, and provides several scaling properties: under stable network conditions, MIP has no control message overhead to maintain multicast routing information; MIP responds to network events as fast as routers can propagate update information, rather than waiting for timers to expire before propagating changes; and finally, because no loops can occur, routers obtain correct multicast routing or stop forwarding data to a portion of a multicast tree as fast as update information can propagate along a desired multicast tree. In addition, as Section 9 illustrates, even when retransmission attempts and topology changes are taken into account, MIP requires less overhead traffic than the "soft-state" approach used in PIM.

The rest of this paper is organized as follows. Section 2 provides an overview of MIP. Sections 3 through 7 describe MIP in detail, provide examples of its operation, and present a formal specification. Section 9 compares the performance of MIP with the performance of PIM.

2 OVERVIEW OF MIP

2.1 Multicast model

Like CBT and PIM, MIP adopts the host group multicast model [4], used in the existing IP architecture. The model defines the service interface to the users of an internetwork. Each multicast address identifies a group of receivers to which a multicast packet is delivered with "best effort." The set of the receivers of a multicast packet is called a host group. To send messages to a group, a sender specifies the destination of the messages with the multicast address of the group. The source does not need to know the addresses of the individual members of the group. Any sender can send to a group, whether or not it is a member of the group. The number and location of members in a group can be arbitrary and dynamic.

IGMP [3] or a similar protocol is assumed for the routers to monitor the presence of group members on their attached subnetworks and to propagate and exchange multicast information. Furthermore, for any local-area network (LAN) with two or more routers, there is a designated router (DR), just as in CBT and PIM, to act on behalf of the end hosts on the LAN to start, join, or end a multicast communication and to transmit communication packets of a group. A simple election mechanism suffices to select the DR, e.g., the router with the largest IP address becomes a DR, or the Hello protocol. However, the specific mechanism to be used is beyond the scope of the paper.

The network consists of an arbitrary interconnection of routers by networks or point-to-point links. A network is represented by a graph $G = (V, E)$, where nodes represent routers, and edges represent links. The links have a time-dependent positive cost. A link is operational if it is operational in both directions. Each router in the network has a link-cost table, which gives the cost of its adjacent links. Each router x knows the next-hop router and the cost metric to destinations from its unicast routing-table URT^x. All messages, link failures,

link recoveries, and link-cost changes are processed one at a time within a finite time and in the order of their occurrence.

2.2 Types of multicast trees

MIP can construct both group-shared (or simply shared) trees and shortest-path trees for multicast routing, and accommodates sender-initiated and receiver-initiated multicast tree construction. These two modes of tree construction are interoperable, e.g., a receiver may join a multicast tree that was sender-initiated. This makes MIP flexible to use with a wide range of applications with different characteristics, group dynamics and sizes.

The shared tree is rooted at a router willing to be the root of the multicast tree. Often, this is a source or receiver router. This is similar to shared trees in CBT or PIM in that a shared tree is rooted at the "core" or "rendezvous point" (RP) router, respectively. The shortest-path trees in MIP can be relaxed to be cost-bounded trees, making it possible to trade off optimality of the tree with the control message overhead of maintaining a shortest-path tree. Just as with PIM, to construct a shortest-path tree, MIP assumes that the link costs are symmetric.

To change the root of the shared tree, in case of failure, MIP employs a ring protocol between the root and all of its neighbors. Only routers on the ring are involved in the changing of the root of the shared tree. CBT and PIM use a ranked list of possible roots, which must be known by all routers on the tree. In CBT and PIM, routers must perform complex procedures to guarantee that the shared tree is rooted at the primary root. The scheme used in MIP is much more dynamic than the ones by CBT and PIM, which use multiple static cores and RPs, respectively.

2.3 Protocol operation

MIP defines the following tree-management computations to create and maintain multicast trees in either a sender-initiated or receiver-initiated mode:

- *Join:* Used by a router not on a multicast tree to become part of the tree.
- *Expand:* In sender-initiated mode, the expand operation allows a router to establish or maintain a multicast tree. In receiver-initiated mode, a router can expand the multicast tree in response to a request from another router to join the tree.
- *Terminate:* Used by a router to tear down its attachment to the tree, as well as the entire subtree below it.
- *Root-update:* Used by a router to update the distance and root information in its multicast subtree in response to changing network conditions.
- *Prune:* Used by a router on a multicast tree to remove itself from the tree.

The names and policies associated with some of the above MIP operations are similar to those defined in PIM (which uses join-prune requests that do not use explicit acknowledgments) and CBT (which uses joins and join acknowledgments). However, the mechanisms used to implement the policies defined for the MIP operations are very different in MIP than in PIM, CBT, and the existing Internet multicast architecture.

PIM is based on the notion of *soft state*, which requires routers in a multicast tree to refresh their membership in the tree periodically (e.g., every 60 seconds, the suggested default value [6]). In contrast, MIP is based on *diffusing computations* [7, 8], which means that every computation started by a router to create and maintain a multicast tree is propagated to other routers as needed using a recursive query-response mechanism. This mechanism allows the router that started a computation to determine when the computation has been completed successfully or if it cannot be completed. More specifically, a router initiates a computation by sending a query to one or more of its neighbor routers and waits for replies from all those neighbors to detect termination of the computation. Each neighbor sends a reply to the query after it terminates the computation, which may require the router to send its own query and receive replies from the corresponding neighbors.

There are three main advantages of implementing tree-management computations in MIP using diffusing computations. First, MIP never has any multicast routing loops. Second, MIP has faster response time, because it does not rely on timers for the periodic transmission of tree membership information or the dissemination of tree changes. Third, being event-driven, MIP incurs no overhead traffic when a multicast routing tree is stable.

A MIP query specifies the computation requested by the sending router from the receiving router and can be an expand, a join, a terminate, or a prune. A reply takes the form of a positive or negative acknowledgement. A query or a reply may have additional information germane to the particular computation requested.

For any given multicast tree, a MIP router can be in different states, and the state of a router in a given tree is independent of its state for any other tree. Accordingly, our description of MIP's operation is with respect to a given tree. A router can change its state when it sends and receives queries and replies, when its adjacent links fail and recover, and when its adjacent hosts join and leave the multicast group for which the tree was built.

A router that is not part of a multicast tree is said to be in the *idle* state. All routers are initialized in the idle state. A router that is part of a multicast tree and is not waiting to complete a computation to establish or maintain the tree is said to be *passive*. A router that is waiting to complete the execution of a computation is said to be *active* for that computation.

Because a router can be active or passive with respect to different computations for a given tree, we label the active or passive state based on the computation that needs to be executed in the tree. Using the expand computation as an example, when an idle router receives an expand query, it enters the expand-active state. When the router performs the necessary computations, it sends a reply. If the reply is a positive acknowledgement, the router becomes part of the tree and enters the expand-passive state. If the reply is a negative acknowledgement, the router does not become part of the tree and goes to the idle state. The other MIP computations have states defined similarly. As a router may be active in multiple different computations, there are complex active states that are formed by the combinations of active states for single computations. Referring to the exact state of a router can thus become cumbersome and overwhelming; therefore, we focus on just one computation at a time in the subsequent description of MIP's operation.

The assumption of a diffusing computation is that a query and its reply travel reliably from one router to its neighbor. However, in an internetwork, MIP would have to run on top of an unreliable datagram delivery service such as that provided by IP or UDP [2]. Accordingly, MIP provides its own retransmissions of queries and replies to ensure that they are exchanged reliably between neighbor routers. When a router receives a query or reply, it sends an acknowledgment of receipt to the sending neighbor, which simply signals the correct reception of the message, not that such a message has been processed.

Each router x stores the information regarding the computation of a multicast tree in its multicast routing table (MRT^x). The incoming and outgoing interfaces of a multicast tree used to forward data packets are also stored in the MRT^x. The information in the MRT^x is indexed by (r, g), where r is the root of a tree for group g. Each entry has a one-bit flag $expand^x$ that indicates if it was established by an expand query or a prune query. If the source s of a data packet matches an expand-established entry (r, g) and arrives on the incoming interface of the entry, the packet is forwarded on the outgoing interfaces of the entry. Otherwise, if the source s of a data packet matches an prune-established entry (r, g) arrives on an adjacent link of the entry, the packet is forwarded on all adjacent tree links except for the one on which the packet was received. If there is no match, the data packet is forwarded using the group-shared tree, which is identified by a one-bit flag $shared^x$ being set for an entry. The shared tree may happen to be the shortest-path of a particular source. If all of the above attempts to match fail, the data packet is discarded. Sources that want to send packets on the shared tree become part of the tree and use it to forward data packets. If a source establishes a shortest-path tree, it sends a prune request to its predecessor on the shared tree in order not to receive it owns packets from the shared tree. On the shared tree, a router forwards a data packet on all adjacent tree links except for the one on which the packet was received. Prune-established entries only exist in routers on the shared tree and are used in forwarding data on the shared tree. They are created by routers which have switched to the shortest-path tree of a source s to prune data packet sent by s from arriving on the shared tree.

The following sections provide a more detailed description of how MIP works using simple cases of its operation. Our description is given with respect to a generic multicast tree for a group g rooted at a source s. Figure 1 specifies the notation used in the formal specification of MIP, presented in Figures 6 to 7 and the following sections. The predecessor of a node in a rooted tree is its parent in the rooted tree. The successors of a node in a rooted tree are its children in the rooted tree. A router in an expand-active state has a predecessor, called expand-predecessor, to whom it owes a reply; and may has successors, called expand-successors, from whom it expects replies. These are defined similarly for the other computations. To avoid cumbersome notation, the subscripting by (s, g) as shown here in the names of auxiliary and state variables and in the routing information associated with a given multicast tree is suppressed in the description and discussion of MIP.

$expand-active$:	a router is expand-active when it has received an expand query and has not yet given an expand reply.
$expand-passive$:	a router is expand-passive when it is on the multicast tree and has no pending expand computation.
$idle$:	a router is idle when it is not on a multicast tree and has no pending computation.
$\delta^x_{s,g}$:	cost slack of x, which the upper bound on the difference between the cost along the multicast tree T and the cost in the internetwork N, i.e., $\|D^{s,x}(T) - D^{s,x}(N)\| < \delta^x$.
$\delta(M)$:	cost slack specified in message M.
$d^{x,y}$:	link cost from router x to y.
$ep^x_{s,g}$:	expand-predecessor of x in (s,g) tree.
$expand^x_{s,g}$:	a one-bit flag indicating if (s,g) was created by an expand computation or a prune computation.
$jp^x_{s,g}$:	join-predecessor of x for (s,g) tree.
$p^x_{s,g}$:	tree-predecessor of x on the path from s.
$q^{x,y}_{s,g}$:	query counter of x for neighbor y.
$q(M)$:	value of counter specified in message M.
$next\_hop(u)$:	next-hop router to destination u.
$r^{x,y}_{s,g}$:	reply counter of x for neighbor y.
$shared^{x,y}_{s,g}$:	a one-bit indicating if (s,g) tree is group-shared.
$D^{x,y}_{s,g}$:	path cost from router x to y in (s,g) tree.
$D^{x,y}(T)$:	path cost from router x to y in graph T.
$D^{x,y}(M)$:	path cost from router x to y in query or reply M.
$DU^x_{s,g}$:	path cost increase update for router x in (s,g) tree.
$EC^x_{s,g}$ or $JC^x_{s,g}$:	active compute expand or join set of x, respectively.
$EC^{x,y}_{s,g}$ or $JC^{x,y}_{s,g}$:	active compute expand or join set of x for neighbor y, respectively.
$ED^x_{s,g}$:	expand done set of x for entry (s,g).
$EQ^{x,y}$:	expand query from x to neighbor y.
$ER^{x,y}$:	expand reply from x to neighbor y.
$ES^x_{s,g}$:	expand-successor set of x.
$JQ^{x,y}$:	join query from x to neighbor y.
$JS^x_{s,g}$:	join-successor set of x.
N^x :	set of neighbors of x in the network.
$PS^x_{s,g}$:	pruner set; the set of neighbors that have sent a prune for (s,g) tree.
$R(M)$:	set of receivers in a query or reply M.
$TS^x_{s,g}$:	tree-successor set of x in (s,g) tree.
UR^x :	set of unreachable destination of router x.

Figure 1: Notation.

3 SENDER-INITIATED TREE CREATION

The sender-initiated tree construction is well-suited for small groups, where it is manageable for the source to know the identity of the receivers. A good example of such an application is a video conferencing session involving only a few sites. This method also gives the source more control over the distribution. The information regarding the identity of the receivers is used by a sender only at the startup to create the tree, and can be discarded thereafter. The information is *not* used by the sender to send data to the group. The sender-initiated tree construction can also be used to start a small group or to create a backbone, and then receiver-initiated tree construction can be used to grow the multicast tree.

The expand computation is the primary means used in the sender-initiated multicast tree creation. In order to keep the description short and simple, the expand computation is presented for constructing a shared tree (Figure 6). The computation proceeds with the transmission and reception of expand queries and replies. In all queries and replies, the source address s and group address g are specified along with other germane information. The source-based multicast tree computation is a recursive generalization of multiple unicast route computations. In addition, the computation is integrated with other computations involved in adapting to network and group membership changes to achieve loop-free multicast routing.

To start a multicast session, a source host notifies its designated router, referred to here as the source router, and provides it with the list of group members. The source router s becomes expand-active for the set of members and sends expand queries towards the group members. The expand queries specify the addresses of the members, each of which may be a host address, a range of addresses, or a router address.

A router receiving an expand query becomes expand-active for the set of members in the query. A router (e.g., source router) uses the next-hop information from the underlying unicast routing table for forwarding queries to group members. If a router receives an expand query that cannot be forwarded because the members in the

query are unreachable, the router replies with a negative acknowledgement (an expand-nack). When a router receives an expand query that specifies a receiver on its subnetwork it replies with a positive acknowledgement(an expand-ack). The replying router includes the receiver address in the expand-ack. The intermediate routers on the paths from the source to the multicast group members propagate the expand-acks back to the source, aggregating the addresses of members for which they have received expand-acks in the replies. The mechanism to prevent loop creation is very simple, and consists of an expand-active router replying with an expand-nack when it receives an expand query for a member for which it is expand-active.

When the source router gets the replies to all its queries, it becomes expand-passive and the multicast tree spanning all reachable members is established. After the source gets all its needed replies, if there are members for which it has not received expand-acks, the source tries again to reach those members. The frequency of retries should allow for an expand to go out and replies to come back. A typical round-trip delay from one corner of the internet to another is on the order of a few hundred milliseconds. This means that the source can retry on the order of a second for a few times (e.g., 3 or 4) before giving up on a receiver.

When s becomes expand-active, it creates an (s, g) entry in MRT^s containing all the pertinent information regarding the multicast tree and its computation, including a bit $expand^s$ set to one for the entry to indicate that it is created by an expand computation. If s becomes the root of the shared tree for a group, it also sets a bit $shared^s$ for the entry. The bit $shared^s$ being set indicates that the particular entry is a shared tree. Assume for simplicity that s is creating a shared tree. Source s sends query $EQ^{s,x}$ to x, which is the next-hop for a subset $EC^{s,x}$ of group members, specifying $EC^{s,x}$, and the cost $D^{s,x}$ (i.e., the link cost $d^{s,x}$). Router s also sets bit *shared* in $EQ^{s,x}$. The bit *shared* in expand queries is used by the routers that receive the queries to set the *shared* bit appropriately for a multicast entry. Each member for which s sends a query is inserted in the set EC^s. Source s designates each neighbor x that receives a query as an expand-successor. Its tree-predecessor and expand-predecessor are set to null. Source s uses a reply counter for each expand-successor to know when it has all the replies from the expand-successor. It increments the counter for an expand-successor by one when it sends a query to the expand-successor. When it receives the reply from the expand-successor, it decrements the counter by a value given in the reply.

As an example, consider the network segment of Figure 2, in which router x is in the idle state. Router x receives an expand query $EQ^{y,x}$ from a neighbor router y in Figure 2a. Router x becomes expand-active for the set of members $\{R1, R2\}$. It creates an entry (s, g) in its multicast routing table MRT^x. It sets its tree-predecessor and expand-predecessor to y. It sets distance measure $D^{s,x}$ to the metric $D^{sx}(EQ^{y,x})$ specified in the query. Router x then performs the same steps as source s to forward expand queries. It forwards a query for $\{R1\}$ and a query for $\{R2\}$, shown in Figure 2b.

While the appropriate queries are being forwarded towards the members, router x receives another expand query $EQ^{z,x}$ from a neighbor router z in Figure 2c. Any number of reasons could have caused $EQ^{z,x}$ to arrive when it does(e.g., unicast routing loop or topology changes). Router x adds the set of members $\{R3\}$ in the query to its expand-active set EC^x. In the example, $D^{sx}(EQ^{z,x}) < D^{s,x}$. As a result, router x sends a negative acknowledgement to y and updates its tree-predecessor and expand-predecessor to z. Furthermore, it forwards an expand query for $\{R3\}$. Meanwhile, expand-acks are propagating back from members $R1$ and $R2$ (Figure 2d). In Figure 2e, the expand-acks for $R1$ reaches x, and the expand query for $R3$ reaches $R3$. The expand-predecessor of $R2$ is waiting for a reply from $R3$. In Figure 2f, the expand-predecessor of $R2$ and $R3$ receives replies to all its queries. In Figure 2g, router x receives an expand-acknowledgement for $\{R2, R3\}$. At that point, router x has received all the replies to its queries. Finally, in Figure 2h, router x sends an expand-ack for $\{R1, R2, R3\}$.

If the network unicast routing tables have not converged to the correct next-hop information, it is possible that some paths to receivers in the multicast tree are suboptimal. Receivers can re-establish paths to meet their optimality requirements using join computations. The details of the mechanism are discussed in Section 5.

4 RECEIVER-INITIATED TREE CREATION

Receiver-initiated tree creation is well suited for groups with a large number of receivers and is based on the join computation. In order to keep the description short and simple, the join computation is presented for constructing a shared tree (Figure 7). A receiver router that wants to become part of a multicast tree sends a join query towards a router on the multicast tree. The query specifies the receiver. When the query reaches a

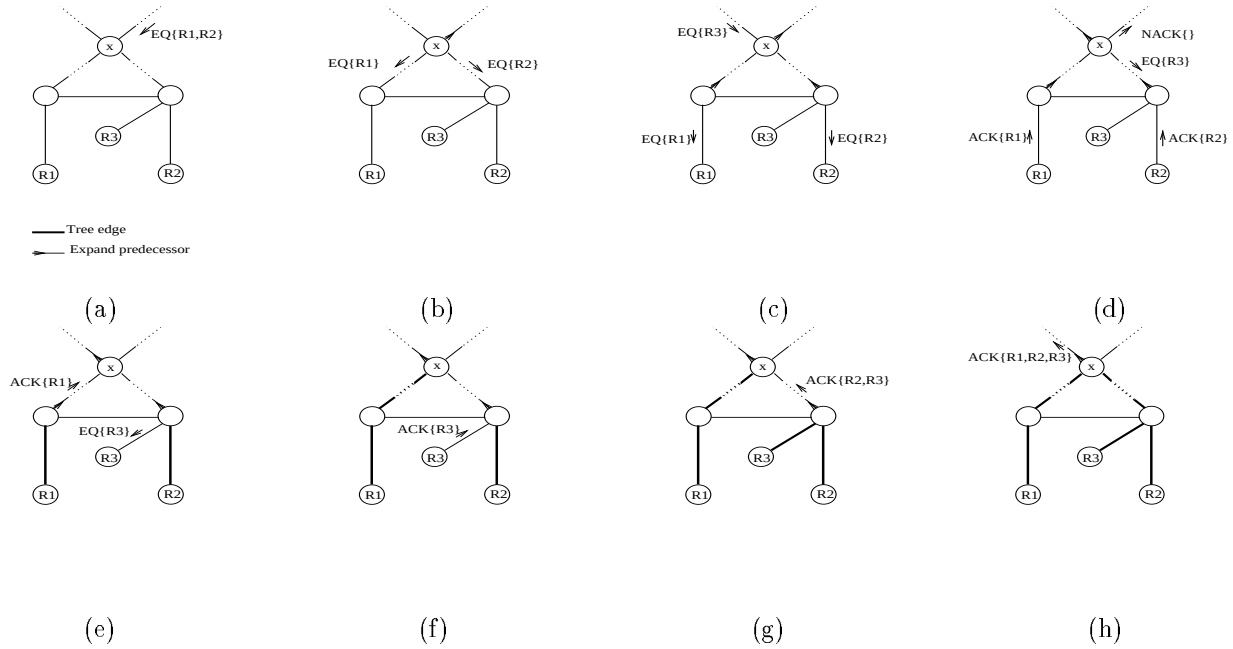


Figure 2: A simple example of expand computation.

router on the multicast tree, an expand query traverses down the path taken by the join query to the receiver, thereby establishing in the multicast tree the path to the receiver.

To simplify the description of the join computation, assume that all routers in the network are in the idle state. In the receiver-initiated tree creation, a receiver needs to know the address of a router on the multicast tree. This is similar to a receiver needing to know the address of a core in CBT or a rendezvous point (RP) in PIM. This information can be obtained when the receivers learn about group addresses. Without loss of generality, let a receiver know the address of the root s of a multicast tree. When an end system receiver wants to join a multicast group, it informs the designated router (DR) on its LAN, using a protocol such as IGMP. (As mentioned before, we have assumed the existence of some election method to pick a DR, e.g., the Hello protocol). In the following, we use receiver to mean the DR of an end system wanting to join a multicast group. Then, the DR x_r acts on the behalf of the end system to become a part of the multicast tree of a group.

To initiate becoming part of the multicast tree, a receiver becomes join-active. While join-active, a router x maintains a set $JC^{x,y}$ for each neighbor y and itself. The set $JC^{x,y}$ contains the set of members for which x has received a join query from neighbor y. The set JC^x, called the join-compute set, is defined as $JC^x = \cup_y JC^{x,y}$. Router x detects looping of a join query when the query contains a receiver in JC^x. Router x initializes the join-successor set $JS^x \leftarrow \emptyset$. The join-successor set contains the set of neighbors from which x has received a join query. Router x initializes its cost metric $D^{s,x} \leftarrow \infty$. Then, router x finds the next-hop h to root s from its unicast routing table URT^x. Router x initializes the join-predecessor to h, and it sends a join query to s via h. It then waits to receive a reply, i.e., an expand message. Finally, when x gets its reply, x sets its tree-predecessor to its join-predecessor and terminates its join computation. Once the join computation terminates at a router, all variables related to the computation are deleted, e.g. $jp^x \leftarrow null$, $JS^x \leftarrow \emptyset$ and $JC^x \leftarrow \emptyset$.

We show the operation of receiver-initiated tree-construction on the network segment of Figure 3. Receiver $R2$ becomes join-active, setting $JC^{R2,R2} \leftarrow \{R2\}$. In Figure 3a, $R2$ sends a join query toward the root of the multicast tree via the next-hop z. Router z becomes join-active, and forwards the join query as shown in Figure 3b, setting $JC^{z,R2} \leftarrow \{R2\}$. Meanwhile, another receiver $R1$ becomes join-active, setting $JC^{R1,R1} \leftarrow \{R1\}$, and sends a join query toward the root via z (Figure 3b). Router z, setting $JC^{z,R1} \leftarrow \{R1\}$, forwards the join query as shown in Figure 3c. Also in the figure, router x becomes join-active for $R2$ and takes the steps given above. Router x propagates a join query for $R2$ towards the root. The join query for $R1$ is forwarded by x toward the root of the multicast tree (Figure 3d). Router x is join-active for $R1$ and $R2$, i.e., $JC^x = \{R1, R2\}$. Now suppose that, due

to unicast routing changes, the join query for $R2$ loops back to router x, as shown in Figure 3e. Router x replies with an join-nack for $R2$ (Figure 3f). A join query for a multicast travels toward the root of the tree until it reaches a router x_i on the multicast tree. Router x_i then sends an expand query with an empty member set, called a basic expand query, to the neighbor from which it received the join query. In the example, a basic expand query arrives at x, as shown in Figure 3g, Router x sets its tree-predecessor to the join-predecessor, forwards the basic expand query to its neighbors in JS^x, and becomes expand-passive (Figure 3h). In Figure 3i, the basic expand queries arrive at the receivers $R1$ and $R2$, at which point they set their tree-predecessor to the join-predecessor, and become idle. The multicast tree established in the process is shown in Figure 3j.

The loop detection at a router x relies on the join-compute set JC^x. Only routers involved in a join computation maintain join-compute sets, and do so only for the duration of the computation. Otherwise, no router on the multicast tree maintains a join-compute set. In practice, the size of join-compute set is expected to be small because, although a join-compute set at a router grows while the router is waiting to be in the multicast tree, the construction of a path in the multicast tree is expected to be fast (on the order of a message propagation to a member of the multicast tree). Moreover, because join queries arrive at a router asynchronously, there is a high likelihood that the portion of the multicast tree (i.e., routers) that needs to be shared by many receivers is established by the first join query from that set of receivers, and that the subsequent join queries from the rest of those receivers will not require the same routers in the shared portion of the tree to start a join computation and become join-active again.

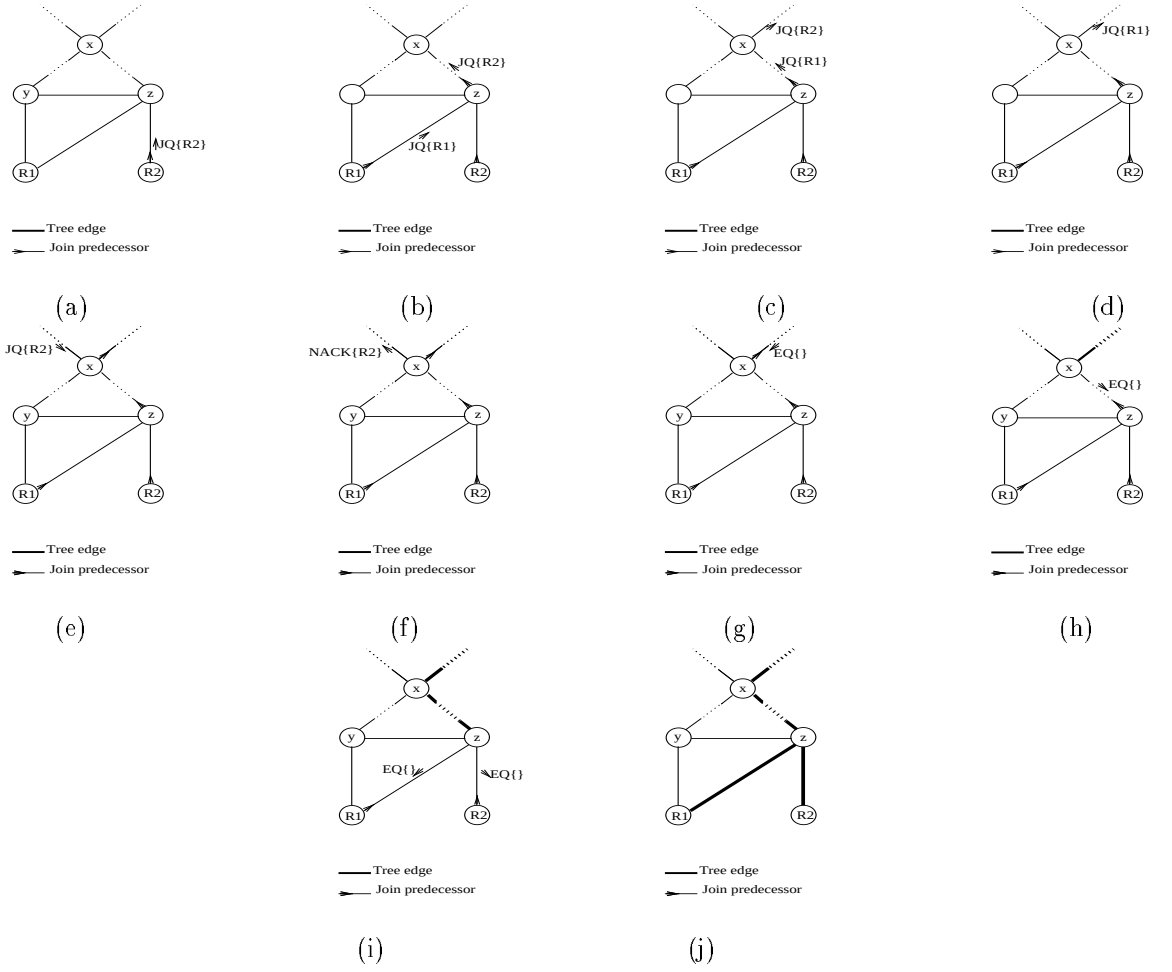


Figure 3: A simple example of join computation.

5 TREE PRUNING

The prune request is used by a router to remove an adjacent link in the multicast tree. This is necessary, for instance, when a member switches to the shortest path tree of a source, or when a leaf member leaves a group.

When receiver x_r on a shared tree with root s^0 wants a shortest-path route from a source s^1, it starts a join computation for s^1, sending a join query towards s^1. In the join query, receiver x_r specifies its cost slack. The cost slack δ^{x_r} is defined to be the upper bound on $|D^{s,x} - D^{s,x}(N)|$, where s is the root of the multicast tree, $D^{s,x}$ is the cost metric along the multicast tree, and $D^{s,x}(N)$ is the cost metric in the internetwork N given by the underlying unicast routing.

A router on the (s^1, g) multicast tree that receives the join query and meets the cost slack of x_r sends a basic expand query to establish the path to x_r. Once x_r becomes part of the multicast tree of s^1, it sends a prune request to tree-predecessor p^{x_r} on the shared tree, specifying s^1 in the request.

Let y be tree-predecessor of x_r on the shared tree, i.e., the (s^0, g) tree. When y receives the prune request, it creates an (s^1, g) entry in MRT^y if it does not have the entry, and sets $expand^y \leftarrow false$ and $shared^y \leftarrow false$ for the entry. The one-bit flag $expand^y$ being false indicates that the forwarding entry was created by a prune request. The one-bit flag $shared^y$ being false indicates that the entry is not a group-shared tree. Router y associates with (s^1, g) a set PS^y, called pruner set, which is initialized with the (s^0, g) tree-successor that sent the prune request, i.e., $PS^y \leftarrow \{x_r\}$. The tree-predecessor of (s^1, g) is set to $p^y_{s^0,g}$. The tree-successor set of (s^1, g) is defined with respect to the tree- and expand-successor sets of the shared tree: $TS^y_{s^1,g} \leftarrow (TS^y_{s^0,g} \cup ES^y_{s^0,g}) - PS^y_{s^1,g}$. The cost metric $D^{s^1,y}_{s^1,g}$ is not used for prune-established entries and is set to ∞. When any of the adjacent links of the shared tree change, the adjacent links of the entries created by prune requests in MRT^y for group g, are updated as above using the new values. If another tree-successor z of y on the shared tree sends a prune request to y for source s^1, router y sets $PS^y_{s^1,g} \leftarrow PS^y_{s^1,g} \cup \{z\}$, and updates $TS^y_{s^1,g}$ as above using the new value of the pruner set.

When y receives a prune request from x_r for the shared tree (s^0, g), i.e., the request specifies s^0, it looks up the group entries in MRT^y created by a prune request from x_r, and removes x_r from the pruner sets of the entries. If a pruner set becomes empty the corresponding entry is removed from MRT^y.

If receiver x_r on an (s, g) multicast tree wants to leave the tree and it does not have any tree-successors, it sends prune request to tree-predecessor p^{x_r} in (s, g) multicast tree. The router p^{x_r} removes x_r from its tree-successor set $TS^{p^{x_r}}_{s,g}$.

6 DYNAMIC SOURCES

If a new source s^n wants to send to an existing multicast group, it joins the shared tree of the group in the same fashion as a receiver. The source sends a join query toward a router on the shared tree. The first router on the shared tree that receives the query takes the necessary steps and sends a basic expand query. The optimality of cost for the path from s^n in the shared tree is not the primary concern in the shared tree. This is also true of the shared trees constructed by CBT and PIM. In CBT and PIM, data packets can experience arbitrarily large costs in the shared tree because the core or the RP can be become poorly placed as a result of failures and recoveries of links and routers in the network. In MIP, as in PIM, those receivers on the shared tree who want optimal costs can switch to the shortest-path tree of s^n.

When a source finishes transmitting to a group, it tears down the part of the routing structures for the group that it solely used. Any source s that is on the shared tree of a group, but not the root of the tree, can leave the shared tree by sending a prune request to its tree-predecessor on the shared tree. If it is the root of a shortest path tree, it sends terminate queries down its tree in order to tear it down.

If the root s of the shared tree wants to leave the tree, it initiates an election protocol on the shared tree to find a new root s'. The election protocol could be, for example, to pick a source on the shared tree with the largest address as the root or a neighbor of the root. After the election, s' sends root-update queries to each neighbor on the shared tree to change all (s, g) entries to (s', g). In the process, the tree-predecessor and path cost for the (s', g) shared tree are set. Each router x that receives a root-update query from y, sends a root-update query to all its neighbors except y on the shared tree. If y is x's only neighbor on the shared tree, it sends a root-update-ack to y. In this way, acknowledgements flow to the new s'. When s gives its reply for a root-update

from s', it sends a prune request to its tree-predecessor on the (s', g) shared tree if it has no tree-successors on the shared tree.

In order to tear-down an (s, g) multicast tree for which a router $x = s$ is the root, x sends terminate queries to its tree- and expand-successors. Once a terminate query reaches a leaf router in the multicast tree, the leaf router replies with a terminate-ack. As the terminate-acks flow back to the root of the tree, the tree is deconstructed. If a router that is in the terminate-active state for a (s, g) tree receives an expand query for (s, g), it sends a negative acknowledgement. If a router s that is no longer the root of a tree receives a join query for the (s, g) multicast tree, it replies with a join-nack.

The failure of the root of a shortest-path tree will result in the tear-down of the tree. For a shared tree, however, it is necessary to find a new root and to re-establish the partitioned tree. This is done by operating a ring protocol between the root s and all of its tree neighbors. Each router in the ring knows all the other routers and their ranking within the ring. This is a bounded size ring in a degree upper-bounded network. Let the members of the ring be indexed by their rank as $\rho^0 = s, \rho^1, ..., \rho^d$, where the root s has the highest rank. If s goes down, $\rho^2, ..., \rho^d$ send join queries to ρ^1. Router ρ^1 also starts an expand computation to reach the rest of the ring. This way ρ^1 knows to update root information in the shared multicast tree when it has all the replies to its expand computation. Each router ρ^i successively tries the next router in rank if it cannot reach some router ρ^j. Router ρ^i does not accept joins to itself if it is trying to join some higher ranking router ρ^j. When router ρ^i has failed to reach all routers ρ^j for $j < i$, ρ^i performs an expand computation to ρ^j for $j > i$, and starts accepting joins from ρ^j. Let ρ^1 be reachable. Once ρ^1 completes its expand computation, it sets up its own ring with its neighbors, and sends a root-update to its tree successors, advertising itself as the new root.

7 DYNAMIC NETWORK CONDITIONS

7.1 Link-cost and distance changes

When a link cost changes, the upstream router x_u of the link sends basic expand queries to each of its tree- and expand-successors. Thus, an expand-passive router becomes expand-active as a result of a link cost change. This way, information about the distance from the root s along the multicast tree is propagated to the affected receivers downstream. Then, each router x in a subtree of x_u updates $D^{s,x}$ when it receives all of its replies and becomes expand-passive. While expand-active, a router stores the latest update of $D^{s,x}$ from the tree-predecessor in $DU^{s,x}$. Each receiver x_r can control the optimality of it path from s along the multicast tree. If at x_r the distance from s along the tree increases beyond a certain tolerance from the shortest-path distance in the network, then x_r sends a join query toward s. Let $D^{sx}(T)$ and $D^{sx}(N)$ be the distance from source s to router x along tree T and in network N, respectively. Receiver x_r can control the suboptimality of cost from s by triggering the establishment of the shortest path when $D^{sx_r}(T) > D^{sx_r}(N) + \delta^{x_r}$ for some positive bound δ^{x_r}.

7.2 Link failure

To avoid deadlock, a router treats the failure of a link as the reception of the (positive or negative) acknowledgement needed from the neighbor(s) on the other side of the link for any computation in which the router is active. Consider a link (x_u, x_d) failing, where x_u and x_d are the upstream and downstream routers, respectively. If $x_d \in TS^{x_u}$, router x_u assumes it has received a prune request from x_d. If x_u is not a member DR and has no tree- or expand-successors, it sends a prune request to its tree-predecessor p^{x_u} and becomes idle. If x_u is expand-active such that $x_d \in ES^{x_u}$, router x_u assumes it has received a expand-nack from x_d, zeroes out the reply counter for x_d, and removes x_d from ES^{x_u}. If x_u is join-active such that $x_d \in JS^{x_u}$, router x_u assumes it has received a join-ack that specifies JC^{x_u,x_d} from x_d. If x_u is terminate-active for the tree, it assumes it has received a terminate-ack from x_d. If x_u is root-update-active, it assumes it has received a root-update-ack.

Responding to a link failure such that tree-predecessor $p^{x_d} = x_u$, router x_d becomes expand-active (if it is not already). If x_d was already expand-active with $ep^{x_d} = x_u$, it zeros out its query counter and sets $ep^{x_d} \leftarrow null$. Router x_d sets $D^{s,x_d} \leftarrow \infty$, and sends basic expand queries to its tree- and expand-successors, specifying an infinite cost from s. When x_d receives all its replies, it tries to join the multicast tree, if it is a member DR or has tree-successors. When trying to join the multicast tree, router x_d sets join-compute set $JC^{x_d,x_d} \leftarrow \{x_d\}$ and join-predecessor $jp^x \leftarrow next\_hop(s)$. The basic expand queries are necessary before the join computation, as a router in the subtree of x_d may have to become an ancestor of x_d to reconnect x_d to the multicast tree. If x_d

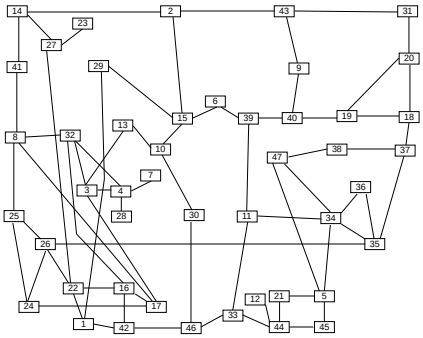


Figure 4: Arpanet.

determines from URT^{x_d} that it cannot reach s, it goes to the terminate-active state and sends terminate queries to its tree-successors to tear-down its subtree. When the terminate computation finishes, x_d becomes idle. If x_d is join-active and cannot reach s, it sends join-nacks to its join-successors, clears the related state information, and becomes idle. If x_d is root-update-active, it finishes it root-update computation and takes the same step as above to join the newly rooted tree. If x_d is terminate-active, it becomes idle when it gets all its replies.

8 MULTIPLE COMPUTATIONS

It can be that there are several computations being carried out for a given tree. If an expand-active router x receives a join query from neighbor y and it satisfies the delay slack, it sends a basic expand query to y and adds y to its tree-successor set TS^x. If x does not satisfy the delay slack, it becomes join-active and send a join query toward the root of the tree. If x is expand-active and receives a prune query y, it removes y from the tree-successor set.

If a join-active router x receives an expand query from a neighbor y for a set of receivers, it becomes expand-active and forwards expand queries as needed. When a join-active router gets an expand query that satisfies its delay slack, it forwards basic expand queries on its join-successors, and terminates its join computation.

If a router x on a shared tree (s^0, g) receives a prune request for a source-specific tree $(s^1, g) \notin MRT^x$, it creates the (s^1, g) entry based on the shared tree (s^0, g) information as described in Section 5. Recall that the flag $expand^x$ for a prune create entry is set to false. When x receives an expand query for (s^1, g) from neighbor y, it becomes expand-active and creates an (s^1, g) entry whose $expand^x$ is set to true. Router x forwards expand queries as needed to establish the (s^1, g) subtree at x.

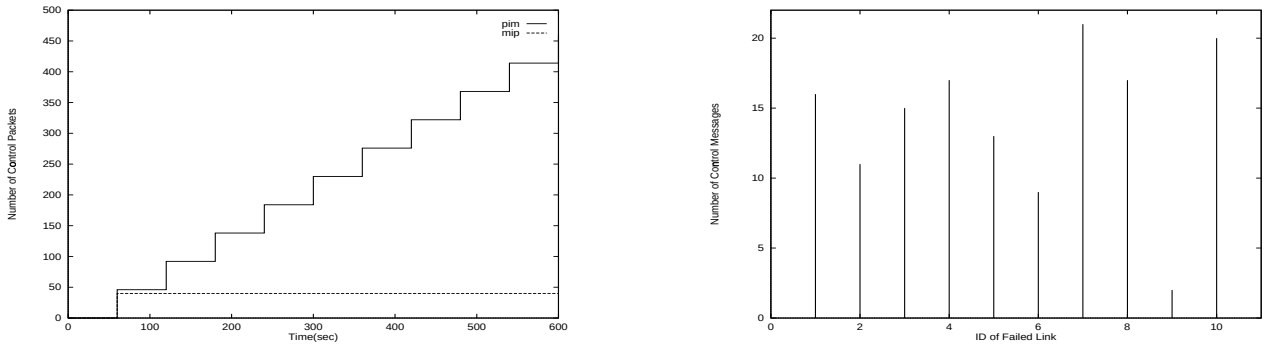
Root-update computation is started only by the new root when it is expand-passive. Any router that is root-update-active sends negative acknowledgements to any other queries that it might receive. Similarly, if a router x has received a terminate query, it does not start any other computation for the multicast tree and it sends negative acknowledgements to any queries (e.g. expand, or join) that it might receive.

9 SIMULATION

We have performed simulations* to evaluate the control message overhead of MIP. For the simulations, we have used the 49-node topology of the Arpanet. The first experiment compares the control message overhead of MIP and PIM for a particular group-shared multicast tree. The multicast group consists of five participant routers: 2, 24, 26, 27 and 29. Each participant acts as both a sender and a receiver. A shared tree is constructed with router 24 being the root and RP of MIP and PIM, respectively. For PIM, the timers are set to 60 seconds, which is the suggested default value for the Join/Prune refresh interval in PIM's specification [6]. For MIP, the receiver-initiated tree creation mode is used. The link delays are assumed to be the same in both directions. The network is lightly loaded, so that the link-delays and processing time is very small (compared to 60 seconds). A Bellman-Ford routing algorithm is the underlying unicast routing protocol. The overhead is measured in terms

*We thank Rooftop Communications for donating the C++ Protocol Toolkit used in our simulation

of the number of control packets, because the sizes of PIM's control packets involved are within a constant factor of the sizes of MIP's control packets. Figure 5 shows the number of control messages to create and maintain the tree. As can be seen, the overhead of MIP is less than that of PIM and it becomes proportionally less and less over time. Making the time-out value smaller makes PIM more responsive to network and group changes,



(a) (b)

Figure 5: (a) Control overhead of MIP vs. PIM. (b)MIP's control overhead for link failures.

but produces more control messages. Making PIM's time-out value longer has the opposite effect, i.e., fewer control messages at the expense of slower adaptation to routing changes. Using this trade-off, control overhead can be fixed to a percentage of the link bandwidths, sometimes referred to as scalable timers. Nevertheless, our experiment illustrates the fact that, in a stable topology, MIP produces far less overhead than PIM, and does so while allowing routers to create the multicast tree as quickly as messages can propagate to the appropriate routers. Note that the overhead of MIP shown includes the cost of hop-by-hop reliable transmission.

The retransmission time-out for reliable hop-by-hop transmission used in MIP would be no faster than the PIM time-out value, with 60 seconds being a good default value. Hence, in a dynamic network with link failures and unicast routing changes, the worst-case number of control messages sent by MIP would be on the same order as the number of messages sent by PIM, and is proportional to the size of the multicast tree, i.e., the number of routers on the tree and not the number of routers in the entire network. This is true because, in the worst case, retransmissions of queries and replies would take place in MIP at the same rate that PIM is forced to periodically refresh its state.

In the second experiment, each link of the above multicast tree was made to fail, and the number of MIP control messages generated to re-establish the tree was counted. The retransmission time-out for reliable hop-by-hop transmission used in MIP is 60 seconds. As can be seen, the overhead in a dynamic network is small, and it is accrued only on an event-triggered basis.

The above experiments are provided only as an illustration of the performance of MIP and PIM in normal operational conditions. In an internet, it is expected that the rate at which resources fail is far smaller than the rate at which messages are exchanged in a multicast tree already established. Accordingly, the design of MIP seeks to minimize overhead traffic while the multicast routing tree is stable, even if additional overhead traffic is needed to modify the state of the tree when network resources fail.

As the results of the two prior experiments illustrate, MIP incurs zero overhead when the multicast tree is established and stable, and incurs similar overhead to that incurred in PIM to establish a multicast tree. This is due to the fact that MIP maintains the state of the multicast tree at every router that belong to the tree, and establishes the tree with operations that are fairly efficient.

The price that must be paid for zero overhead during periods of stability is that, when links of the multicast tree fail, portions of the multicast tree may have to be explicitly modified or flushed, depending on whether or not there are alternate paths to reach the multicast tree. In the worst case, the number of messages that MIP requires to change or flush a subtree with x nodes is $O(4x)$, because an expand operation and its acknowledgments must propagate down and up the tree to erase old cost information, and a second operation and its acknowledgments must propagate in the same subtree to either modify or flush the subtree. In contrast, PIM requires routers to send joins periodically towards the RPs and the RPs to propagate reachability messages to the receivers, which takes $O(2N)$ messages.

However, because topology changes affecting the multicast routing tree should be rare (e.g., occur less frequently than once every few minutes), flushing or modifying an entire subtree is a rare event, specially for large portions of the multicast routing tree. Therefore, incurring $O(4x)$ messages after failures of links of the multicast tree is much preferable to incurring $O(2N)$ or $O(V)$ messages periodically as in PIM or DVMRP, where N and V are the number of nodes in the multicast tree and network, respectively. MIP's savings over PIM and DVMRP becomes more significant in long-lived multicast sessions.

10 CONCLUSION

We have presented a new multicast routing protocol MIP, which solves the shortcomings identified in the current IP architecture, PIM, and CBT. MIP offers a flexible and simple approach to the construction of group-shared and shortest-path multicast trees. It is easy to accommodate the needs of a wide range of multicast applications from sparse widely-distributed replicated databases to delay-sensitive interactive multimedia applications. Additionally, MIP can be sender-initiated or receiver-initiated. Therefore, it can be better tailored to the particular nature of an application's group dynamics and size. For instance, in the receiver-initiated mode, MIP can readily accommodate traditional IP multicast architecture. MIP delegates the maintenance of path delays to the receivers, who can tradeoff the control message overhead with the cost optimality of their paths. MIP is independent of the underlying unicast routing algorithms, and is robust and adapts under dynamic network conditions, such as topology or link cost changes. MIP has a fast response time to network conditions since network events such as link failures are propagated as fast as messages can travel, as opposed to timers expiring to reflect the new state. Under stable network conditions, MIP has no control message overhead for tree maintenance.

ACKNOWLEDGEMENTS

This work was supported in part by the Advanced Research Projects Agency under contract F19628-93-C-0175.

References

[1] A. J. Ballardie, P. F. Francis, and J. Crowcroft. Core-based Trees (CBT). In *Proc. of SIGCOMM '93*, pages 85–95, 1993.

[2] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1992.

[3] S. E. Deering. Host Extensions for IP Multicasting. *RFC 1112*, Aug. 1988.

[4] S. E. Deering and D. Cheriton. Multicast Routing in Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8:85–110, May 1990.

[5] S. E. Deering et al. The PIM Architecture for Wide-area Multicast Routing. *IEEE/ACM Transactions on Networking*, 4(2):153–162, April 1996.

[6] Stephen Deering et al. Protocol Independent Multicast-Sparse Mode (PIM-SIM): Protocol Specification. *Internet Draft*, Sept. 1995.

[7] E. W. Dijkstra and C. S. Scholten. Termination Detection for Diffusing Computation. *Inform. Process. Lett.*, 11(1):1–4, 1980.

[8] J. J. Garcia-Luna-Aceves. Loop-Free Routing Using Diffusing Computations. *IEEE/ACM Trans. on Networking*, 1(1):130–141, 1993.

[9] J. Moy. Multicast Extension to OSPF. *Internet Draft 1584*, 1994.

[10] M. Parsa and J.J. Garcia-Luna-Aceves. Scalable Internet Multicast Routing. In *Proc. of ICCCN '95*, pages 162–166, 1995.

[11] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. *RFC 1075*, Nov. 1988.

```
[This procedure is executed by router x when x
receives an expand query EQ^{y,x} from router y.]
Procedure ExpandQuery(EQ^{y,x})
begin
  if jp^x ≠ null
    if jp^x ≠ y and |D^{s,x}(N) − D^{s,x}(EQ^{y,x})| < δ^x
      send join-ack(x, jp^x, JC^x);
      jp^x ← null;
      delete all join-related information;
    endif
  endif
  if p^x = null
    if R(EQ^{y,x}) = ∅
      return;
    endif
    create a multicast tree entry in MRT^x;
    expand^x ← true;
    shared^x ← true;
    p^x ← y;
    D^x ← D^{s,x}(EQ^{y,x});
  endif
  if ep^x = null
    ep^x ← y;
    q^x ← 0;
    ED^x ← ∅;
  endif
  call ForwardExpandQuery(EQ^{y,x});
  if y = p^x
    DU ← D^{s,x}(EQ^{y,x}))
  endif
  q^x ← q^x + 1;
  if (D^{s,x} > D^{s,x}(EQ^{y,x}))
    if ep^x ≠ null
      send expand-nack(x, ep^x, q^x, ∅);
    endif
    if p^x ≠ ep^x
      send prune(x, p^x, {r});
    endif
    p^x ← ep^x ← y;
    q^x ← 1;
  else
    if ep^x ≠ null
      send expand-nack(x, y, q^x, ∅);
    endif
  endif
  if ES^x = ∅
    call SendExpandReply();
  endif
end
```

```
[This procedure is executed by router x when x
receives an expand reply ER^{y,x} from router y.]
Procedure ExpandReply(ER^{y,x})
begin
  ED^x ← ED^x ∪ R(ER^{y,x});
  EC^x ← EC^x − R(ER^{y,x});
  r^{x,y} ← r^{x,y} − q(ER^{y,x});
  if r^{x,y} = 0
    ES^x ← ES^x − {y};
    if ER^{y,x} is expand − ack
      TS^x ← TS^x ∪ {y};
    endif
  endif
  if ES^x = ∅
    if DU^{s,x} ≠ D^{s,x}
      D^{s,x} ← DU^{s,x};
    endif
    if D^{s,x} = ∞ and p^x = jp^x = null
      if receiver^x = true or TS^x ≠ ∅
        JC^x ← JC^{x,x} ← {x};
        δ^x ← ∞;
        jp^x ← next_hop(r);
        send join(x, jp^x, JC^x);
        return
      endif
    else
      call SendExpandReply();
    endif
  endif
end
```

```
Procedure ForwardExpandQuery(EQ^{y,x})
Local Variable: X^x, R^x, ES*^x
begin
  X^x ← {u | u ∈ R(EQ^{y,x}) and u ∉ EC^x};
  R^x ← {u | u ∈ X^x and u is reachable};
  if R^x ≠ ∅
    ES*^x ← ∅;
    [update expand-compute sets and find next-hops]
    foreach u ∈ R^x
      if next_hop(u) ≠ null and next_hop(u) ∉ (ES^x ∪ TS^x)
        ES*^x ← ES*^x ∪ {next_hop(u)};
        EC^{x,next_hop(u)} ← EC^{x,next_hop(u)} ∪ {u};
      endif
    endforeach
    [forward expand queries]
    foreach n ∈ ES*^x
      if EC^{x,n} ≠ ∅
        r^{x,n} ← r^{x,n} + 1;
        send expand(x, n, DU^{s,x} + d^{x,n}, EC^{x,n});
      endif
    endforeach
  endif
  [forward basic expand queries]
  ES^x ← ES^x ∪ TS^x;
  foreach n ∈ (ES^x − ES*^x)
    r^{x,n} ← r^{x,n} + 1;
    send expand(x, n, DU^{s,x} + d^{x,n}, ∅);
  endforeach
  ES^x ← ES^x ∪ ES*^x;
  if |DU^{s,x} − D^{s,x}(N)| < δ^x
    foreach n ∈ JS^x
      send expand(x, n, DU^{s,x} + d^{x,n}, ∅);
    endforeach
    JS^x ← ∅;
    if ES^x = ∅
      delete all tree-related information;
    endif
  endif
  EC^x ← EC^x ∪ R^x;
end
```

```
Procedure SendExpandReply()
begin
  if p^x ≠ ep^x
    send expand-nack(x, ep^x, q^x, ED^x);
  else
    if ED^x ≠ ∅ or receiver^x = true or TS^x ≠ ∅
      send expand-ack(x, ep^x, q^x, ED^x);
    else
      send expand-nack(x, ep^x, q^x, ED^x);
      p^x ← null;
    endif
  endif
  delete all expand-related information;
  if p^x = null and jp^x = null
    delete all tree-related information;
  endif
end
```

Figure 6: Expand computation for constructing a shared tree for a group.

```
[This procedure is called when node x receives an
join query from node y for multicast tree (s, g). ]
Procedure JoinQuery(JQ^{y,x})
begin
  if p^x = null and jp^x = null
    D^{s,x} ← ∞;
    JS^x ← {y};
    JC^x ← JC^{x,y} ← R(JQ^{y,x});
    δ^x ← δ(JQ^{y,x});
    jp^x ← next_hop(s);
    if jp^x ∉ JS^x
      send join(x, jp^x, JC^x);
    else
      send join-nack(x, y, JC^{x,y});
    endif
  else if p^x = null and jp^x ≠ null
    L^x ← R(JQ^{y,x}) ∩ JC^x;
    if L^x ≠ ∅
      send join-nack(x, y, L^x);
    else
      JS^x ← JS^x ∪ {y};
      JC^{x,y} ← JC^{x,y} ∪ R(JQ^{y,x});
      JC^x ← JC^x ∪ R(JQ^{y,x});
      δ^x ← min(δ^x, δ(JQ^{y,x}))
      send join(x, jp^x, R(JQ^{y,x}));
    endif
  else if p^x ≠ null
    if DU^{s,x} − D^{s,x}(N) < δ(JQ^{y,x})
      send expand(x, y, DU^{s,x} + d^{x,y}, shared^x, ∅);
    else
      if jp^x ≠ null
        if δ(JQ^{y,x}) < δ^x and next_hop(s) ≠ jp^x
          send join-ack(x, jp^x, JC^x);
          jp^x ← next_hop(s);
        endif
        if jp^x ∈ JS^x
          send join-nack(x, jp^x, JC^{x,jp^x});
          JC^{x,jp^x} ← ∅;
        endif
        JS^x ← JS^x ∪ {y};
        JC^{x,y} ← JC^{x,y} ∪ R(JQ^{y,x});
        JC^x ← JC^x ∪ R(JQ^{y,x});
        δ^x ← min(δ^x, δ(JQ^{y,x}))
        send join(x, jp^x, JC^x);
      endif
    endif
  endif
end

[This procedure is called when node x receives an
join reply from node y for multicast tree (r, g).]
Procedure JoinReply(JR^{y,x})
begin
  if JR^{y,x} is join-nack
    JC^x ← JC^x − R(JR^{y,x});
    if JC^x = ∅
      foreach n ∈ JS^x
        send join-nack(x, n, JC^{x,n});
      endforeach
      delete all join-related information;
    else
      foreach m ∈ R(JR^{y,x})
        foreach n ∈ JS^x
          if m ∈ JS^{x,n}
            send join-nack(x, n, {m});
          endif
        endforeach
      endforeach
    endif
  else
    if y ∈ JS^x
      JC^{x,y} ← JC^{x,y} − R(JR^{y,x});
      JC^x ← JC^x − R(JR^{y,x});
      send join-ack(x, jp^x, R(JR^{y,x}));
      if JC^{x,y} = ∅
        JS^x ← JS^x − {y};
      endif
      if JC^x = ∅
        delete all join-related information;
      endif
    endif
  endif
end
```

(a)

```
[This procedure is called when the link (x, y) fails]
Procedure LinkDown(x, y)
begin
  foreach tree (s, g) ∈ MRT^x
    if (p^x = y)
      p^x ← null;
      DU^{s,x} ← ∞;
      q^{x,y} ← 0;
      [send basic expand queries]
      EQ^{z,x} ← expand(z, x, ∞, shared^x, ∅);
        call ForwardExpandQuery(EQ^{z,x});
    endif
    if (jp^x = y)
      [send join query towards s]
      jp^x ← next_hop(s);
      send join(x, jp^x, JC^x);
    endif
    if y ∈ ES^x
      [assume receiving an expand-nack]
      ER^{y,x} ← expand-nack(y, x, ∅, r^{x,y});
      call ExpandReply(ER^{y,x});
    else if y ∈ TS^x
      [assume receiving a prune request]
      PQ^{y,x} ← prune(y, x, {s});
      process prune query PQ^{y,x};
    endif
    if y ∈ JS^x
      [assume receiving a join-ack]
      JR^{y,x} ← join-ack(y, x, JC^{x,y}, null);
      call JoinReply(JR^{y,x});
    endif
  endforeach
end

[This procedure is called when the link (x, y)
has a new cost d.]
Procedure LinkChange(x, y, d)
  d^{x,y} ← d;
  foreach tree (s, g) ∈ MRT
    EQ^{x,y} ← expand(x, y, D^{s,x} + d, shared^x, ∅);
    call ForwardExpandQuery(EQ^{x,y});
  endforeach
end
```

(b)

Figure 7: (a) Join computation for constructing a shared tree for a group. (b) Link-down and cost-change procedures.